Preprint
UCRL-JC-151791

Enhancing Scalability of Parallel Structured AMR Calculations

A. M. Wissink, D. Hysom, R. D. Hornung,

This article was submitted to 17th Annual Association of Computing Machinery International Conference on Supercomputing, San Francisco, CA USA, 6/23/2003 – 6/26/2003

February 10, 2003

This work was performed under the auspices of the United States Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Enhancing Scalability of Parallel Structured AMR Calculations*

Andrew M. Wissink, David Hysom, Richard D. Hornung
Center for Applied Scienti£c Computing
Lawrence Livermore National Laboratory
[awissink,hysom,hornung]@llnl.gov

ABSTRACT

This paper discusses parallel scaling performance of large scale parallel structured adaptive mesh refinement (SAMR) calculations in SAMRAI. Previous work revealed that poor scaling qualities in the adaptive gridding operations in SAMR calculations cause them to become dominant for cases run on up to 512 processors. This work describes algorithms we have developed to enhance the efficiency of the adaptive gridding operations. Performance of the algorithms is evaluated for two adaptive benchmarks run on up 512 processors of an IBM SP system.

1. INTRODUCTION

Structured adaptive mesh refinement (SAMR) [2, 3] is an effective technique for focusing computational resources in numerical simulations of partial differential equations that span a range of disparate length and time scales [5, 6]. AMR is used to dynamically increase grid resolution locally to resolve important fine-scale features in the solution. The goal is to achieve a more efficient computation than one in which a globally-uniform fine grid is applied. SAMR is a particular brand of adaptive mesh refinement technology in which the locally-refined grid is defined with structured grid components. Like other dynamic mesh refinement approaches, SAMR presents complications for parallel computing that are absent in uniform grid calculations. Complex data communication arises from the need to transfer data between grid regions of differing resolution on irregular locally-refined grid configurations. Since grid generation may be performed frequently, the complexity of computing grid-dependent data exchange information cannot be amortized over an entire calculation. Also, substantial data transfers may occur as the grid is refined and coarsened.

*This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. UCRL-XX-XXXX.

The SAMRAI (Structured Adaptive Mesh Refinement Application Infrastructure) [12, 11, 13] provides parallel SAMR infrastructure support for a number of multi-physics applications [7, 10]. It provides general adaptive meshing and data management capabilities as well as a flexible algorithm development framework. In this paper we focus on the performance of parallel calculations with SAMRAI. Specifically, inefficiencies that arise when problem sizes are scaled up to large numbers of processors and algorithms that we have adopted to enhance their efficiency. We present results for simple benchmark applications that use the same solution technique and algorithms as are used in more complex multi-physics applications. Hence, the efficiency enhancement techniques described in this paper apply generally to all applications implemented in SAMRAI, and should be applicable to other SAMR applications as well.

This paper begins by presenting background information about the characteristics of the adaptive problems solved in this paper. We highlight certain sources of inefficiency and introduce algorithms developed to enhance performance. Lastly, we present performance results of adaptive calculations performed using the new algorithms. We provide a breakdown of computational costs in benchmark calculations run on up to 512 processors of ASCI IBM Blue Pacific and on a large-scale Linux cluster.

2. SAMR BACKGROUND

The basic features of the SAMR approach are rooted in the work of Berger, Oliger, and Colella [2, 3]. The computational grid consists of a collection of structured grid components, organized into a hierarchy of nested levels of spatial (and often temporal) grid resolution. Each level in the hierarchy represents a domain with uniform grid spacing. The domain on each level is expressed as a disjoint union of logically-rectangular "patch" regions (see Fig. 1). A finer level is constructed by selecting cells that require refinement and clustering these cells into a new set of patches. Boundary conditions on the finer level patches are obtained from interpolated data on the next coarser level in the hierarchy, or from neighboring patches with the same mesh spacing. As the solution evolves, it is necessary to move, resize, or remove finer level patches as dictated by the needs of the computation.

In the SAMR approach, the refinement factor is constrained to be an integer so that meshes on all levels align. Finer level patches are rectangular and are always nested within

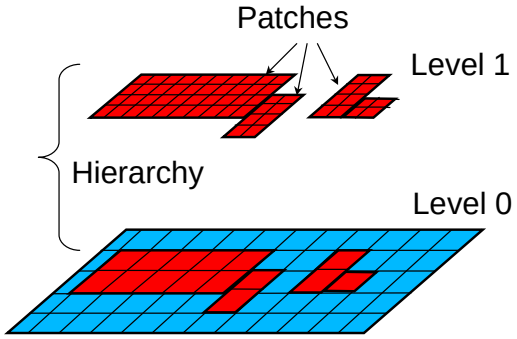

Figure 1: Breakdown of computational mesh in SAMRAI.

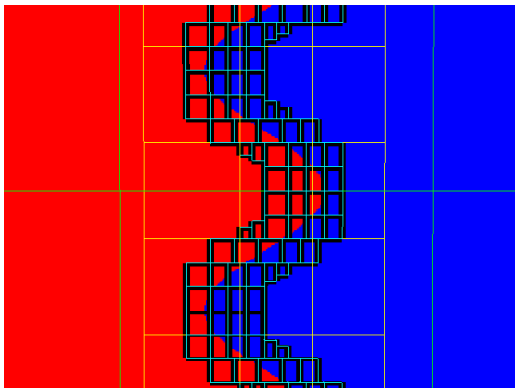

Figure 2: Fine patches resolving discontinuity. When resolving a feature in a SAMR calculation, it is typical to use many smaller patches rather than a few large patches because the refined region can be more efficiently placed around the feature.

the domain of the next coarser level. All levels in the hierarchy are stored, so that fine levels are stored in addition to, not instead of, the overlapped coarser level. This may at first seem wasteful, but it is generally the case that finer levels require much more storage than coarser (e.g. consider a typical three-dimensional problem with a refine ratio of 4; every refined coarse grid cell will be covered by 64 fine grid cells in the refined region). The overhead is consequently quite small. The reduction in the number of gridcells in the adaptive method, relative to its uniform-grid counterpart, is highest when the refined region represents a small percentage of the overall domain of the problem. In most problems, the phenomenon resolved through grid refinement is a complex non-rectangular shape that is most-efficiently captured using a number of smaller patches, as opposed to a single refined large patch block (see Fig. 2).

3. SAMRAI

SAMR offers potentially large savings in memory and computational effort when compared to globally-uniform static mesh calculations. However, difficulties associated with its implementation often make the application of SAMR prohibitive. Apart from the development of numerical methods for locally-refined grids, it is important to keep in mind the complexity of data management and communication on parallel systems. Data must be exchanged among irregularly configured patch regions on a single level and between patches on different levels of resolution. These data communication patterns change whenever the grid changes. Data management becomes more complex in multi-physics applications. Such problems typically involve many data quantities with different centerings on the grid (cell-centered, node-centered, etc.), irregular data such as particles, and different solution procedures that share variables and use distinct data communication patterns.

The SAMRAI framework facilitates parallel multi-physics SAMR applications by providing software tools to manage the complex data exchanges. The object-oriented software design used in SAMRAI captures the salient features of data communication in SAMR applications in a general framework, enabling extensible and specializable high-level algorithm components as well as allowing application developers to specialize operations for their needs. Further discussion of algorithmic design components of SAMRAI is given in ref [12]. The communication infrastructure is discussed in ref [13].

The framework has been used for a variety of parallel multi-physics applications. Dorr, Garaizar, and Hittinger [7] use SAMRAI to develop an adaptive simulation of laser-plasma instabilities. Hornung and Garcia [10, 8] have developed a "hybrid" simulation capability in SAMRAI which uses SAMR to couple an Eulerian fluid model to a Direct Simulation Monte Carlo (DSMC) [1] particle model to model complex fluid interface dynamics, such as the Richtmeyer-Meshkov instability. Anderson and Pember have used SAMRAI to couple Arbitrary Lagrangian-Eulerian hydrodynamics solution techniques with the Adaptive Mesh Refinement for higher-fidelity resolution of shock hydrodynamics applications. In each of these cases, SAMRAI provides high-level solution algorithm components as well as the parallel communication infrastructure.

In this paper, we do not focus on the performance of a single application. Rather, we construct simple benchmarks that exercise the algorithms fundamental to *all* applications implemented in SAMRAI. Of course, each application may have its own particular parallel performance issues. But it is fair to say that any application implemented in SAMRAI should benefit from the performance improvements we explore herein.

4. APPLICATION BENCHMARKS

We evaluate SAMRAI performance for standard SAMR applications that use the well-known explicit hydrodynamics algorithm of Berger and Colella [2]. The algorithm uses both spatial and temporal mesh refinement during time integration and applies numerical flux correction operations to maintain global conservation across multiple refinement levels. Object-oriented features of the SAMRAI implementation of this algorithm and its use in other applications are described elsewhere [12]. Here, we focus on aspects relevant to parallel performance.

4.1 SAMR Algorithm

During time advancement, re-meshing operations are interleaved with integration steps. During re-meshing, computa-

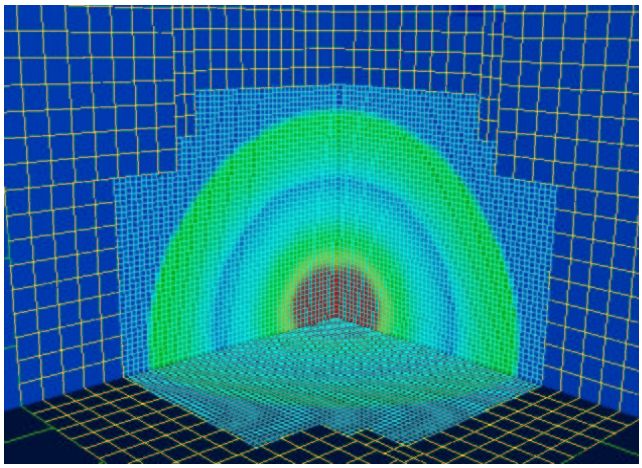

Figure 3: Density contours overlaid on adaptive grid system for spherical shock calculation.

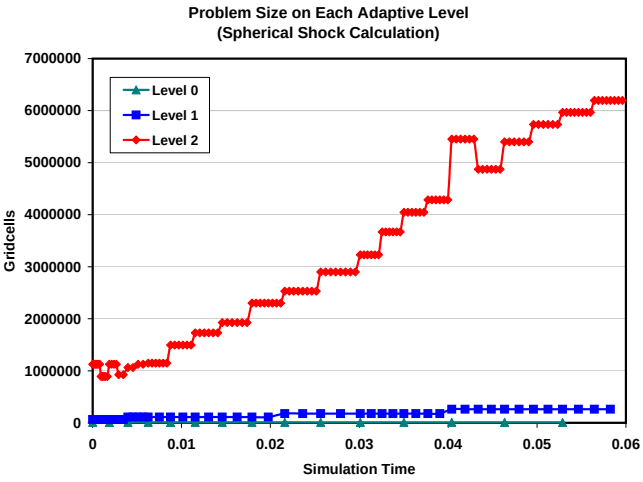

Figure 4: The number of cells on the finest level grows during the coarse of the simulation. The number of cells on the two coarser levels remains roughly the same.

tional cells are selected to identify regions where refinement is needed. Our load balance approach distributes sets of patches to different processors. Cell-tagging is performed on each patch separately and is therefore quite scalable. We use the signature pattern recognition algorithm of Berger and Rigoutsos [4] to cluster tagged cells into logically-rectangular patch regions. Note that re-meshing and integration operations are performed one level at a time. Thus, each level is load balanced separately from the others.

The Berger-Rigoutsos algorithm implementation in SAMRAI performs parallel array reductions over the irregular grid structure to build tag signature arrays on each processor. The accumulation of the tagged cells into signature arrays uses global all-reduce operations. These collective communication operations synchronize the procedure on each processor so that the processors construct identical box regions from which to build new patches. This implementation was intended for small numbers of processors where the cost of global all-reduce operations are negligible.

The boxes constructed by the Berger-Rigoutsos algorithm are further massaged for load balancing. For example, a large box may be chopped into a set of smaller boxes to make it easier to assign the boxes to processors. Each box is chopped until its size is less than the per-processor average size, computed by dividing the total number of computational cells by number of processors. Boxes are then ordered according to their spatial location using a Morton space filling curve algorithm [9] which places a curve through the box centers and partitions the curve. The goal of this last step is to maximize the assignment of adjacent patches to the same processor. The boxes assigned to each processor are used to generate patches on that processor.

Once a new patch level is constructed and load balanced, the integration algorithm constructs new communication schedules for the new patch configuration. The use of communication schedules in SAMRAI is discussed in detail in [13]. Briefly, a schedule is a list of transactions that must take place to satisfy filling data in certain regions of patches. For example, fine level patches may need to interpolate data from a coarser level in the hierarchy, or exchange data with neighboring patches of the same refinement. The communication schedule holds a list of transactions for each patch local to its processor which defines 1) the region of data that must be updated, and 2) the patch that will supply this data. The patch supplying the data may be either local or located on another processor. The list of transactions maintained by the schedule is used to build asynchronous MPI message streams that are exchanged when communication is invoked. For a particular grid configuration, a schedule, once constructed, may be used over and over to exchange data in the grid hierarchy. But when the grid configuration changes, and a new set of patches is constructed, the schedule must be reconstructed.

We investigate parallel performance of two adaptive problems implemented using SAMRAI. The first models a 3D propagating spherical shock with the Euler equations of gas dynamics. This problem is not scaled; that is, the same sized problem is run on all processor partitions. The second problem models a 3D sinusoidal advecting front with the scalar linear advection equation. This problem is scaled, in that the problem size increases proportionately with number of processors. We use the non-scaled problem to investigate how SAMR calculation perform when processors are added to a fixed problem. We use the scaled problem to study trends as problem size is increased as more processors are applied.

Both problems employ the hyperbolic time integration algorithm supplied by SAMRAI. They differ only in the number of variables involved and the operations performed in the numerical kernels. One solution variable appears in the linear advection application while a system of five variables represents the solution in the Euler case. Because the computational effort to update the numerical solution in the linear advection case is less, the linear advection problem has higher data communication and re-meshing costs relative to total computation time than the Euler case.

4.2 Non-scaled Problem

The adaptive Euler hydrodynamics problem is solved using three levels of mesh resolution where the mesh is refined by a factor of four between successive levels; see Figure 3. Figure 4 shows how the number of computational cells on each level changes with simulation time. The number of cells on the finest level constitutes 94% – 96% of the total cells in the calculation. Of the total time spent in the time integration portion of the solution process, 97% – 98% is spent on the finest level for all processor partitions. Problem size grows roughly linearly as the simulation advances during the course of the 15 coarsest grid timesteps over which we ran the computation. This growth is due from the fine mesh adapting to resolve the spherically-expanding shock. The same problem size is used on all processor partitions.

The two primary phases of the calculation are grid generation and time integration. Grid generation involves three major steps: construction of new patch regions from tagged cells (Berger-Rigoutsos procedure plus load balance), construction of communication schedules, and data movement from the old mesh configuration to the new configuration. Time integration uses numerical routines outside of SAMRAI. Data movement to fill ghost cell regions during time integration and to redistribute data during re-meshing is performed by SAMRAI.

4.3 Scaled Problem

Scaled parallel speedup studies are performed with an advecting sinusoidal front problem. In this experiment, we control the mesh generation process by manually scaling the mesh so that the number of gridcells per processor remains constant for the duration of each computation. To do this, we first run a problem on P processors and store the mesh after each re-meshing step. This mesh is then refined as we increase the number of processors. To go from P to $2P$ processors, we double the number of cells in one direction. To go from $2P$ to $4P$, we double the cells in an additional direction, and so on. The use of the linear advection equation allows us to force the same time-stepping sequence on all refined grids. The use of a sinusoidal front helps us make grid configurations more representative of typical SAMR problems.

The adaptive problem uses three levels of mesh resolution where the mesh is refined by a factor of four between levels, as shown in Figure 5. Figure 6 shows how the number of computational cells on each level changes with simulation time. As was the case with the non-scaled spherical shock problem, the vast majority of cells are on the finest refinement level. However, note that the overall problem size remains roughly constant over the course of the computation. The calculation is run over a total of 25 coarsest grid timesteps.

5. ADAPTIVE GRIDDING PERFORMANCE

The two primary phases of the calculation are grid generation and time integration. Previous results [13] for the benchmarks described in section 4 revealed scaling inefficiencies primarily in the grid generation phase of the calculation. Grid generation involves three major steps: construction of new patch regions from tagged cells (Berger-Rigoutsos procedure plus load balance), construction of communication

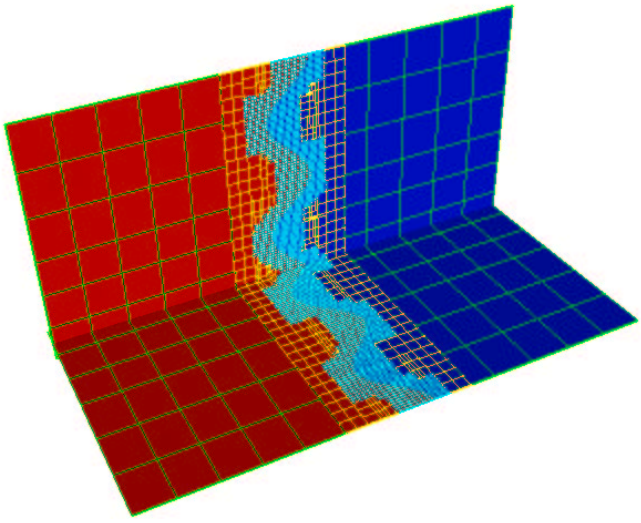

Figure 5: Scaled advecting sinusoidal front problem - density contours overlaid on adaptive grid.

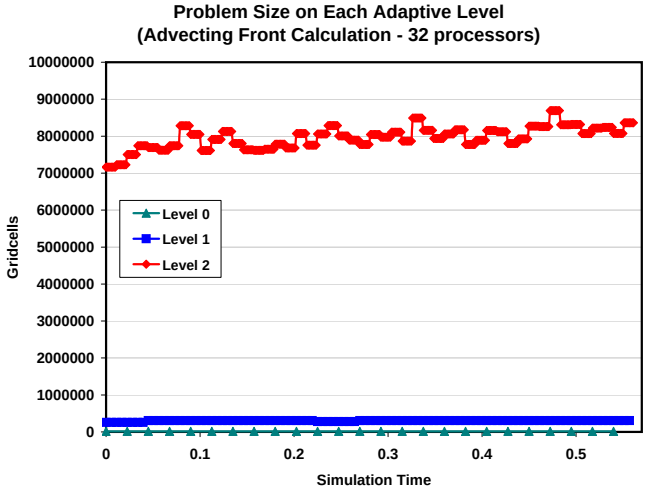

Figure 6: The number of computational cells on each level for the 32 processor case of the scaled advecting front calculation. Unlike the non-scaled case, the problem size stays roughly constant. For the 64, 128, 256, and 512 processor cases, the pattern is identical but the number of cells increases proportionally (e.g. 64 processor case has 2X cells of the 32 processor case, etc.).

schedules, and data movement from the old mesh configuration to the new configuration. One may suspect the most inefficient part of the grid generation process on parallel systems is the data redistribution step. However, we found this step is quite fast, requiring less than 1% of the total execution time on the spectrum of processors tested. Also, the data redistribution phase scaled reasonably well as we increased the number of processors. The primary sources of inefficiency were actually the Berger-Rigoutsos clustering and schedule construction phases. These two phases showed negative scaling characteristics. That is, their total parallel processing time actually *increased* with increasing number of processors.

In our original tests with a scaled adaptive benchmark, we found that as the problem size increases proportionally to the number of processors, the cost to construct communication schedules also grows with the problem size. For smaller problems, the communication schedule construction cost was reasonably small. For example, in the sinusoidal front calculation performed on 32 processors, the cost to construct schedules is 14% of the total time. But as the problem is scaled up to run on more processors, the time to construct the communication schedules rapidly becomes the dominant cost in the calculation, due to poor scaling qualities.

The second scaling inefficiency was in the Berger-Rigoutsos algorithm. For the fixed-size spherical shock benchmark, we found the cost of this phase of the calculation increased. Consider that because the problem size is fixed, even with no scaling the wallclock time should remain constant with increasing number of processors. The fact that it is *increasing* clearly indicates a scaling problem! As with the communication schedule costs, Berger-Rigoutsos trivial on smaller numbers of processors but its poor scaling causes it to becomes non-trivial on larger numbers. In the spherical shock problem, Berger-Rigoutsos used less than 1% of the total time on 32 processors but used 26% of the time on 512 processors.

6. MODIFIED ALGORITHMS

Previous work, summarized above, showed that communication schedule construction costs and the cost of performing Berger-Rigoutsos clustering can have a large impact on SAMR performance when running large problems on large numbers of processors. There are some basic strategies that can be employed to lessen these costs within the existing algorithm. First, one may reduce the re-gridding interval, thereby reducing the number of times the inefficient operations are called. The drawback with this approach is that it is necessary to employ a larger refined region to contain important flow features within the refined level, and avoid it moving into coarser levels where solution fidelity will be lost. Second, codes can be tuned to generate fewer, larger patches rather than many smaller patches. This is advantageous since our experience was that the cost of schedule construction grows roughly as $O(N^2)$, where N is the total number of patches used in the problem. Unfortunately, optimal load balancing is most efficient when the problem has many smaller patches so there arises a tradeoff in the optimization strategies for load balance and communication schedule generation cost.

In this work, we investigate alternative algorithms that can enhance the efficiency of the re-gridding operations without having to resort to modification of the problem itself to improve efficiency.

6.1 Communication Schedules

The original implementation of the schedule construction algorithm used an approach that compared the patch bounding indices of all patches to one another. For example, in order to fill ghost regions of a patch, we computed so-called *box intersections* (a box is simply the index space of the patch) by comparing each patch against all others in the problem. Boxes are represented by very simple data structures; the only information one has is the logical Cartesian coordinates of upper and lower corners. This $O(N^2)$ cost algorithm (N is the total number of patches in the problem) worked well as long as N remained reasonably small. However, N tends to grow proportionately with the number of processors so when the problem size is scaled to large parallel systems, the cost of this algorithm becomes prohibitive.

We propose two alternatives, discussed in the following subsections, to replace the $O(N^2)$ algorithms with algorithms of lower complexity. Although difficult to analyze precisely, in practice the new algorithms appear to have asymptotic running times of $O(N^{3/2})$ and $O(N \log N)$.

6.1.1 BoxGraph algorithm

In the first approach, we developed a graph theoretic model of the box-intersection problem. We theorized a graph, $G(V, E)$, whose vertex set V contains a vertex corresponding to every box in the problem, and whose edgeset E contains an edge (i, j) if and only if the boxes corresponding to vertices i and j intersect. If we could construct such a graph efficiently in practice, then solving the box intersection problem becomes trivial: to find all boxes that intersect a given box, one simply examines the adjacency list of the corresponding vertex.

The *BoxGraph algorithm* we designed for constructing a **BoxGraph** is summarized below. The algorithm operates on a type of divide-and-conquer approach. The $O(N)$ boxes are partitioned into $\sqrt{N}$ subsets; the naive algorithm is applied to each subset to form a collection of subgraphs; and the subgraphs are assembled to form the global graph.

ConstructBoxGraph(boxlist B, boxlist C)

0. Form set S to be the union of B and C.
1. Select a sorting key and sort the boxes in S.
2. Assuming S contains n elements, divide the sorted list into $\sqrt{n}$ bins.
3. Apply the naive algorithm to form a subgraph in each bin.
4. Union the subgraphs to form the BoxGraph.

This approach requires that, to the largest extent possible, boxes within a subset intersect only with other boxes in the same subset. We use a heuristic box-sorting method to attempt to enforce this constraint. Boxes are sorted by picking a sort direction, i.e., x, y, or z with respect to the

Cartesian coordinate system, then sorting boxes by the coordinate of a specified edge. For example, we can sort boxes by placing them in non-decreasing order with respect to the x-coordinate of their left-hand edges. Once sorted, partitioning is easily performed, e.g., by throwing the left-most $\sqrt{N}$ boxes in the first subset, the next $\sqrt{N}$ boxes in the second subset, and so on. The BoxGraph construction algorithm is illustrated in Figure 7.

Analytically it can be shown that, in the worst case (e.g., every box intersects at least one box in every subset) BoxGraph construction can incur a runtime cost as high as $O(N^{5/2})$, which is worse than the naive approach. In practice, however, we have found that construction cost always appears to perform $O(N^{3/2})$.

6.1.2 BoxTop algorithm

In our second approach we also perform a preprocessing phase that consists of sorting boxes. In this approach we sort boxes in every possible direction, e.g., in 2D we would construct four arrays, and sort the boxes by left-hand side, right-hand side, top, and bottom. During a box-intersection operation we use these arrays in conjunction with binary search to obtain a (hopefully small) subset of boxes with which a box of interest may intersect.

The arrays are used per the following example. The coordinate of the right-hand side of a box of interest is compared against the array of boxes that are sorted by their left-hand sides. A box $b[j]$ in the array can only intersect with the box of interest if the coordinate of $b[j]$'s left-hand side is less than the box of interest's right-hand side. Binary search, which takes $0(\sqrt{N})$ time, is used to locate the box (if any) that meets this criteria and for which j is as large as possible. The boxes in array positions $1..j$ then form a "candidate" subset with which the box of interest may intersect. However, boxes in this subset are not guaranteed to intersect with the box of interest.

In 2D we process all four arrays (six arrays in 3D), and choose the smallest subset. We then apply the naive algorithm against every box in the chosen subset. This gives the exact subset of boxes with which the box of interest intersects.

6.1.3 Comparison of BoxTop and BoxGraph

Both BoxTop and BoxGraph methods were used in a revision of our schedule construction algorithms. The BoxGraph construction preprocessing phase requires as input two lists of boxes, e.g., a list of boxes from a level j, and a list of boxes from a level $j-1$ from which the ghost regions on level j are to be filled. In contrast, the BoxTop preprocessing phase requires a single list of boxes as input, and can hence be performed, during level construction. Hence, there are places in the code where it is more "natural" to use BoxTop, and other places where BoxGraph is more appropriate.

As illustrated in Figure 8, both methods are in practice much faster than the naive approach. Runtime for the BoxTop preprocessing phase is bounded below by $\Omega(N^{3/2})$. This is a lower bound since, as explained above, if there are many intersections between boxes in different subsets, the complexity can be as high as $O(N^{5/2})$. After construction, find-

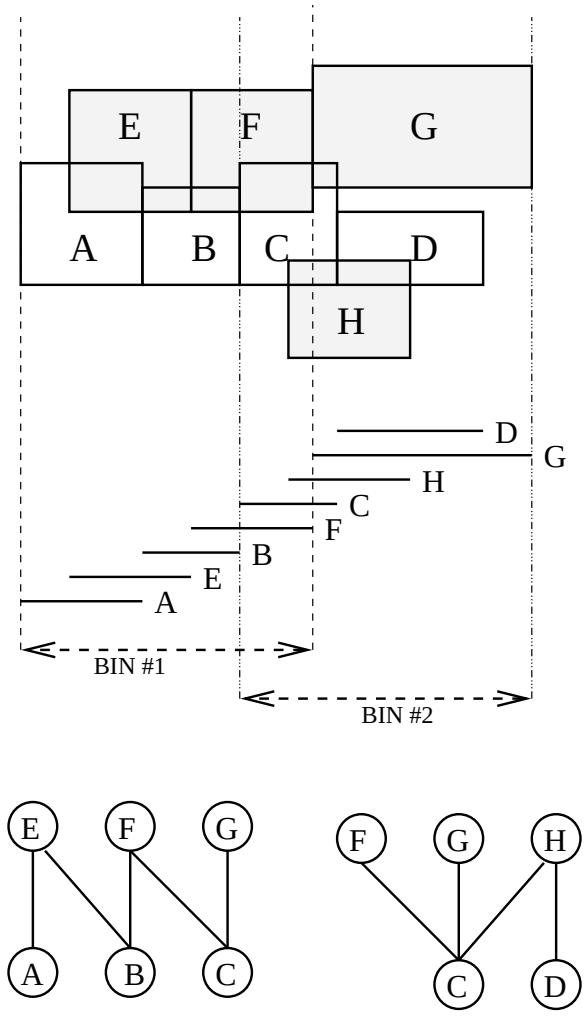

Figure 7: BoxGraph algorithm, phase 1. Top: two sets of boxes, $B = \{A, B, C, D\}$ and $C = \{E, F, G, H\}$. Middle: the line segments that correspond to each box, and their division into bins; the segments are assumed to have been sorted by their left-hand endpoints; boxes A, E, B, F, C and H are assigned to bin 1, and boxes F, C, H, G, and D are assigned to bin 2. Bottom: the subgraphs that are computed for the bins; the complete graph is the union of the two subgraphs.

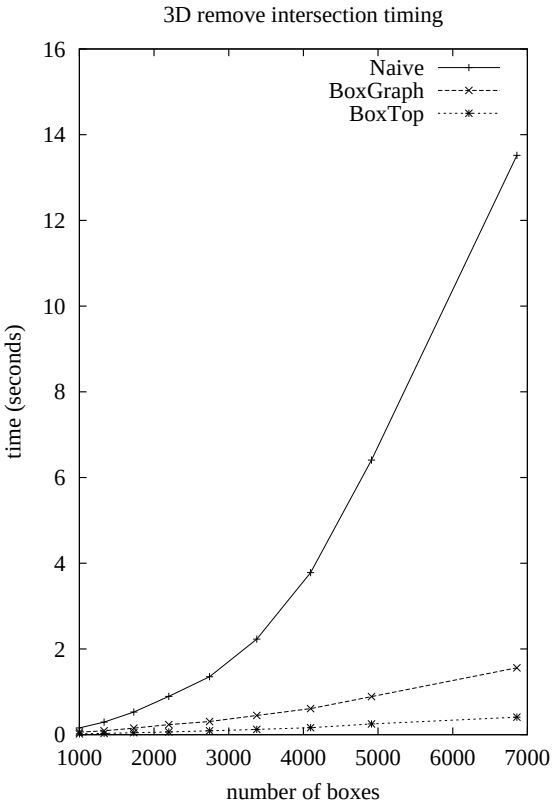

Figure 8: Comparison of BoxGraph, BoxTop, and Naive algorithm on a collection of irregularly tiled boxes in 3D.

ing the boxes that intersect any given box is an 0(1) operation. The BoxGraph preprocessing phase is bounded by $\Omega(N \log N)$, which is a well-known bound on sorting. After construction, finding the boxes that intersect any given box is bounded below by $\Omega(\log N)$. $O(\log N)$ is the time required for binary search; this is a lower bound, however, since there is no guarantee that the smallest subset will be smaller than N.

6.2 Berger-Rigoutsos

The signature pattern recognition algorithm of Berger and Rigoutsos [4] is used during regridding to cluster tagged cells into logically-rectangular patch regions. The algorithm proceeds as follows:

BR_Cluster(box_in, boxlist_out)

1. Compute signature arrays in each direction. (see Figure 9)
2. Compute Gaussian (2^{nd} derivative) of the signature.
3. Compute an inflection point where Gaussian changes most rapidly. (Either along the x or y direction.)
4. Split **box_in** into two new boxes at inflection point.
5. For each of the two new boxes:

5a. Append the new box to **boxlist_out**.

5b. If efficiency constraints are not met, recurse.

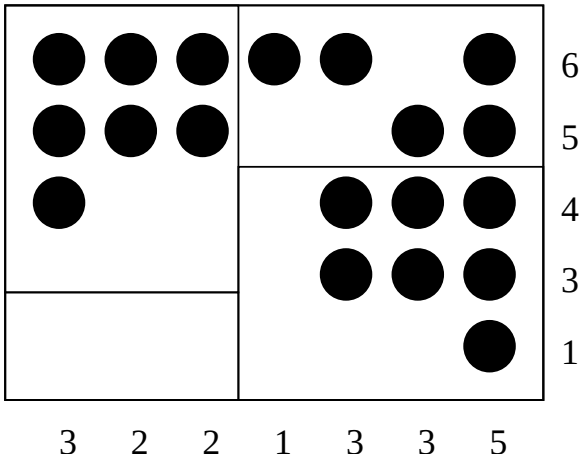

Figure 9: Computation of signature arrays. A 2D rectangular domain consisting of four patches is shown. Darkened circles indicate tagged cells. The signature arrays are the numbers along the bottom and right side; they indicate the number of tagged cells in each column and row.

The input is a box that encompasses all tagged cells. The output is a list of generated boxes. An example *efficiency constraint* is that the ratio of untagged to tagged cells within a box be less than some specified value.

In the SAMRAI implementation, each processor has a list of all boxes on a given level. However, the data associated with a particular box (i.e., the "patches") are distributed across processors. Computational cells that are tagged to identify regions where refinement is needed is performed local to each processor, on the patches it owns, so cell-tagging is quite scalable. The inefficiency arises in the accumulation of tagged cells into signature arrays. Our original implementation used global all-reduce operations to form the signature arrays on each processor (Step 1 of the BR_Cluster algorithm). For the remaining steps, processors use this data in an identical fashion to partition the box into logically rectangular regions, which are the foundation of the next set of patches.

Our intuition was that, on average, only a small number of processors actually need to participate in signature array computations. Processors that do not own any patches that intersect the box currently being split (at some level of recursion) have no information to add to the signature arrays. Moreover, such a processor does not need to participate in any deeper recursive call. If we could avoid having these processors participate in the all-reduce operations, then we might be able to greatly reduce the collective communication costs.

To test this idea we instrumented our code to record various statistics during the Berger-Rigoutsos operation. The results, which are summarized in Table 1 confirmed our suspicion that, during the majority of the recursions, only a small subset of processors had relevant tagged cell data. For example, referencing the line for the 5th recursion level, we

Table 1: Berger-Rigoutsos participation for a 5-step Euler sphere computation on 128 processors. There were a total of 588 calls to Berger-Rigoutsos.

Recursion Level	Participating processors	Percent of total calls
11	2	1.4
10	4	7.1
9	8	17
8	8	30.6
7	8	46.3
6	8	61.9
5	18	74.5
4	32	84.7
3	80	91.5
2	80	95.6
1	100	98
0	125	100

see that 74.5% of all calls to Berger-Rigoutsos required 18 or fewer processors.

These observations led to a reformulation of our Berger-Rigoutsos implementation, which is summarized as follows:

BR_Cluster_Mod1(box_in, boxlist_out)

1. If this processor has no box that intersects with box_in and this is not the root processor, then return.
2. Form communicator with remaining processors.
3. Compute signature arrays in each direction, using all-to-one reduction, with the result stored on the root processor.
4. If this is the root processor:

4a. Compute Gaussian (2^{nd} derivative) of the signature.

4b. Compute an inflection point where Gaussian changes most rapidly.

4c. Split **box_in** into two new boxes at inflection point.

4d. Append the new boxes to **boxlist_out**.

5. Root processor broadcasts the two new boxes to all processors in the current communicator.
6. For each of the two new boxes:

6a. If efficiency constraints are not met, recurse.

7. If this is the outermost recursion level, the root processor broadcasts **boxlist_out** to all other processors.

In addition to having a list of all boxes on a given level, each processor has an array that maps every box to the processor that owns it. This information is used in Step 2. If there are p processors in the current communicator that cumulatively own m boxes, then determining which processors participate in the new communicator can be accomplished in $O(m)$ time. Note that, in Step 11, the contents of **boxlist_out** after the recursion returns are really only meaningful to the root processor, since other processors will have partial lists.

7. PERFORMANCE RESULTS

We compare performance of the algorithms described in section 6 used for the application benchmarks described in section 4. The benchmarks are run on the IBM ASCI Blue Pacific system, which is constructed of 256 four processor SMP nodes, 244 of which are available for typical batch runs. Each processor is a 332 MHz PowerPC 604e. Each node has 1.5 GB memory. An omega topology interconnect network supports up to 150Mbytes/s bi-directional bandwidth between nodes.

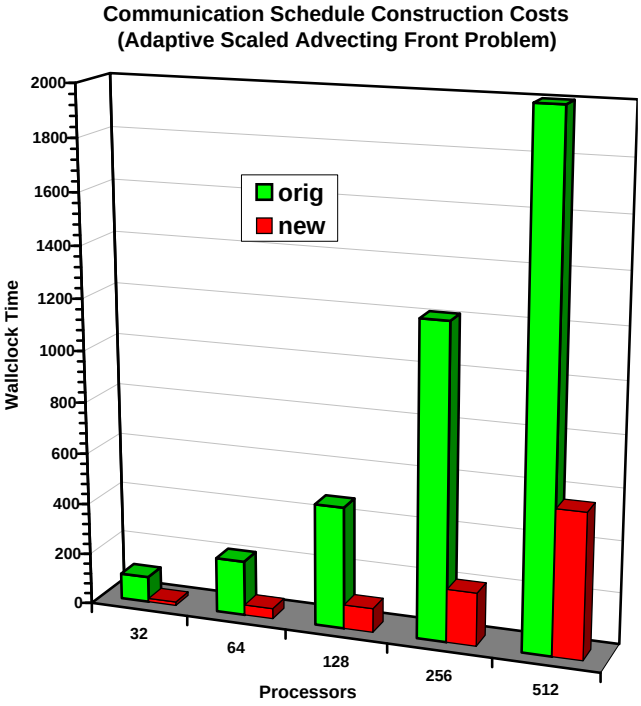

Figure 10: Cost to generate communication schedules in the scaled advection front problem. The column labeled "orig" was measured with our original implementation, while "new" is the modified implementation using coupled BoxGraph-BoxTop algorithms to compute box intersections.

7.1 Modi£ed Algorithm Performance

The time to construct communication schedules using the new BoxGraph-BoxTop algorithms was measured for the scaled scalar advection benchmark. Figure 10 shows the schedule construction time for the benchmark using our original algorithm and the new BoxGraph-BoxTop algorithm. Clearly, the new algorithm shows considerable improvement over our original implementation.

We next compare the time to construct new level boxes from clustered cells using our new parallel implementation of the Berger-Rigoutsos algorithm. Figure 11 shows the time to perform the operation using our original implementation compared to the new implementation that uses a binary-tree reduction for the non-scaled spherical shock benchmark. The improved parallel implementation of Berger-Rigoutsos is considerably faster than our original implementation.

7.2 Non-scaled Benchmark Performance

This section discusses the performance measured for the adaptive spherical shock benchmark discussed in section 4.2.

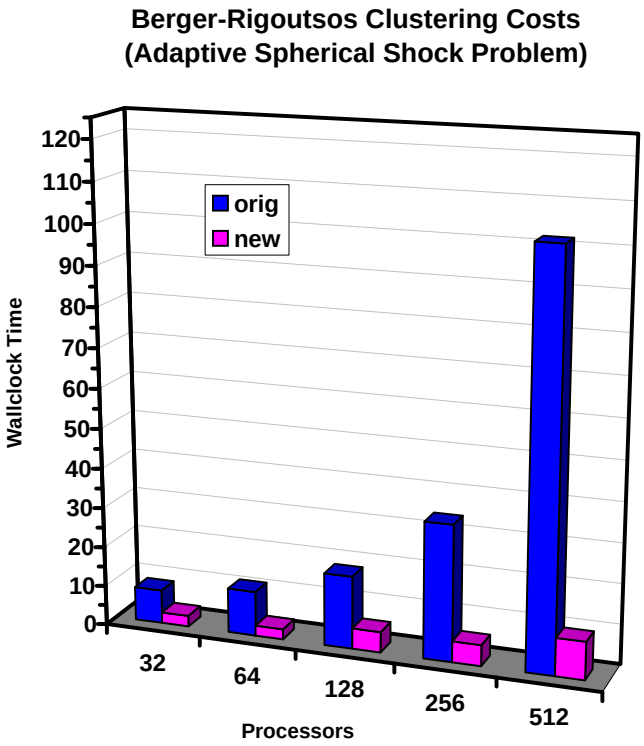

Figure 11: Cost to perform parallel Berger-Rigoutsos clustering operations in the non-scaled spherical front problem. The column labeled "orig" was measured with our original implementation that invoked all-reduce operations during clustering, while "new" is the modified implementation that invokes a binary-tree reduction algorithm.

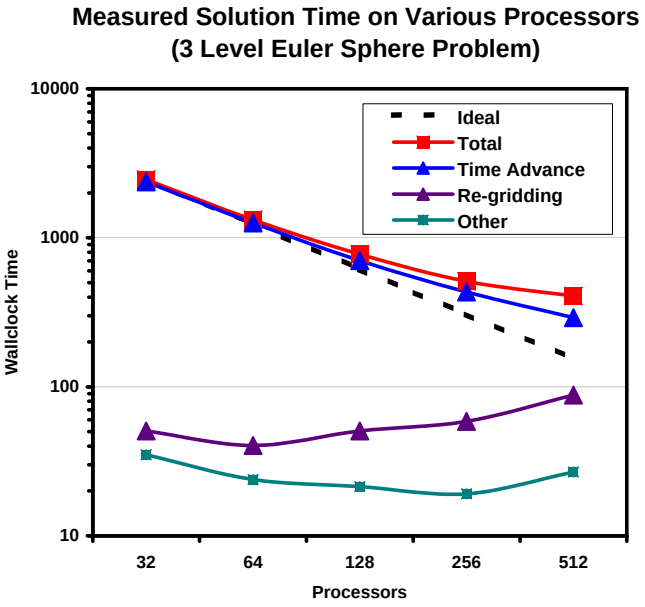

Figure 12: Wallclock time measurements from Table 2 for non-scaled spherical shock calculation.

Table 2 shows the timings of the non-scaled spherical shock benchmark run on the IBM. Wallclock times measured on different processor partitions are reported. Timings are decomposed into the time integration and grid generation phases. The entry labeled "other" includes parts of the calculation that fell outside the time integration and grid generation phases, such as cell-tagging, level data initialization, and load balancing. The breakdown of time integration and regridding is also shown in the plot in Fig. 12.

We expect to see some reduction in parallel efficiency because the same problem is run on all processor partitions, hence the problem size per processor *decreases* as the number of processors is scaled up while collective communication costs necessarily *increase*. In spite of this, we still see reasonable parallel scaling for this adaptive problem. Note that the time spent in Berger-Rigoutsos is minimal on all processor partitions. The schedule construction is still the most costly phase of the regridding operations but still represents a reasonably small percentage of the total cost.

7.3 Scaled Problem Performance

The improvements in the schedule construction costs allow us to achieve reasonably good scaling efficiency for this adaptive problem. Table 3 shows the measured wallclock timings of the scaled advecting sinusoidal front benchmark. The breakdown of time integration and regridding is also shown in the plot in Fig. 13.

The time integration phase of the calculation is scaling quite well. Note that communication costs, including both communication of ghost cells during time integration and data redistribution during regridding, constitute a fairly small percentage of the total costs. The regridding phase of the calculation shows poorer scaling than the operations in the time integration phase algorithm, but is considerably better than results we reported previously. Regridding constitutes

Processors	32		64		128		256		512	
Total	**2458.3**		**1320.5**		**773.9**		**511.3**		**405.7**	
Time Integration	**2372.8**	**97%**	**1256.3**	**95%**	**701.8**	**91%**	**433.6**	**85%**	**290.9**	**72%**
Computation - num kernels	2208.1	90%	1160.1	88%	636.7	82%	388.2	76%	258.8	64%
Communication overhead	164.7	7%	96.2	7%	65.1	8%	45.4	9%	32.1	8%
Grid Generation	**50.5**	**2%**	**40.3**	**3%**	**50.6**	**7%**	**58.5**	**11%**	**88.0**	**21%**
Schedule construction	35.7	2%	30.9	2%	41.5	5%	50.8	10%	76.8	19%
Berger Rigoutsos	2.8	0%	2.6	0%	5.2	1%	5.3	1%	9.5	2%
Data re-distribution	12.0	1%	6.8	1%	3.9	1%	2.5	1%	1.8	1%
Other	**35.0**	**1%**	**23.9**	**2%**	**21.4**	**3%**	**19.1**	**4%**	**26.8**	**7%**

Table 2: Timing results for non-scaled spherical shock calculation. Results show wallclock time for each operation and percentage of total wallclock time for the different phases of the calculation.

Processors	32		64		128		256		512	
Total	**509.7**		**549.9**		**630.0**		**780.7**		**1226.3**	
Time Integration	**468.0**	**92%**	**482.1**	**88%**	**497.9**	**79%**	**509.1**	**65%**	**552.8**	**45%**
Computation - num kernels	461.3	91%	472.0	86%	485.0	77%	498.0	64%	520.6	43%
Communication overhead	6.7	1%	10.1	2%	12.9	2%	11.0	1%	32.1	3%
Grid Generation	**15.0**	**3%**	**41.4**	**8%**	**94.2**	**15%**	**207.5**	**27%**	**563.6**	**46%**
Schedule construction	13.9	3%	40.3	7%	93.1	15%	206.5	26%	562.5	46%
Data re-distribution	1.1	0%	1.1	0%	1.1	0%	1.1	0%	1.1	0%
Other	**26.7**	**5%**	**26.4**	**5%**	**37.9**	**6%**	**64.1**	**8%**	**109.9**	**9%**

Table 3: Timing results for scaled advecting front calculation. Results show wallclock time for each operation and percentage of total wallclock time for the two main phases of the calculation, time integration and grid generation.

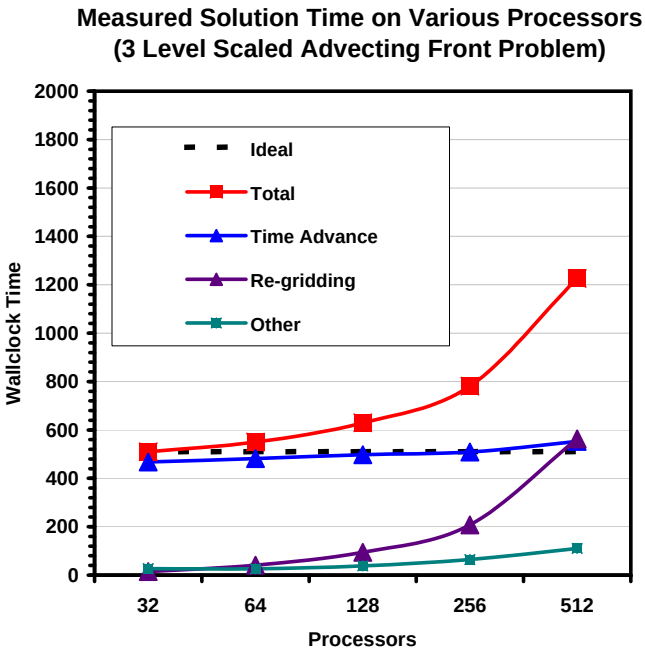

Figure 13: Wallclock time measurements from Table 3 for scaled advecting front calculation.

a small percentage of the overall time except on the largest processor partitions.

A consideration that should be noted is that this scalar advection benchmark involves very simple numerical kernels and solves only a single flow variable Hence, the time integration costs in this benchmark are relatively low compared to a typical adaptive multi-physics application implemented in SAMRAI. On the other hand, regridding costs depend primarily on box operations and not on numerical computations so, for the same dynamic mesh configuration, regridding costs for this benchmark will be comparable to more numerically intensive multi-physics applications. We therefore expect that in practice, regridding will constitute a lower proportion of the total time than what this benchmark demonstrates.

8. CONCLUDING REMARKS

This paper investigates algorithms to enhance the scaling efficiency of structured adaptive mesh refinement (SAMR) applications. In results reported in earlier work [13] we showed that the core numerical kernel operations scaled quite well. Communication constituted a relatively small percentage ($<$ 10%) of the total time. The primary source of scaling inefficiency on large processor counts was adaptive gridding operations. That is, the operations associated with dynamically evolving the mesh to track the feature of interest. For cases run on less than 64 processors, the standard algorithms used in adaptive gridding generally worked well. They constituted a sufficiently small percentage of time that any lack of scaling efficiency could be ignored. However, as larger problems were run on larger numbers of processors, poor

scaling efficiency caused of adaptive gridding costs to dominate. We introduce algorithms to enhance the efficiency of adaptive gridding and demonstrate their performance on two adaptive benchmarks run on up to 512 processors.

Communication schedules, which describe the dependencies between patches on different levels of refinement, must be reconstructed each time the grid changes in a dynamically adaptive problem. The algorithm we originally used in constructing communication schedules was of $O(N^2)$ complexity, where N is the total number of patches in the problem. As the problem size grows, the number of patches also grows and at some point the cost becomes prohibitive. We introduce algorithms which reduce the cost to $O(N \log N)$ and $O(N^{3/2})$. In practice, this reduces the time to construct communication schedules to the point that the cost is comparable to the numerical kernels for a scaled adaptive problem on 512 processors. The modified algorithms are significantly better than our previous algorithm, for which the communication schedule construction far outweighed all other costs with this case.

The process of constructing new patches from regions tagged for refinement was another source of inefficiency. We replaced our original parallel implementation of the Berger-Rigoutsos algorithm, which used global reductions, with an implementation that uses a binary-tree reduction algorithm. In an adaptive benchmark resolving a moving spherical shock, the new implementation reduced the time of this operation from 23% of the total with the original implementation to only 2% with the modified implementation.

In the scaling studies performed in this work on up to 512 processors, indications are that the core numerical operations and communication should continue to scale to larger numbers of processors. On the other hand, adaptive gridding costs begin to become more dominant on larger processor partitions. Although we were able to make significant progress in reducing their costs through the use of more efficient algorithms, further enhancements will likely be necessary as we push to larger numbers of processors.

9. FINAL PAPER

We are currently evaluating performance of our adaptive benchmarks on a linux cluster system, containing 1152 dual-processor 2.4GHz Intel nodes with a high-speed Quadrics switch that has a sustained transfer rate of 220 MB/s and $< 5\mu s$ latency. Although we were not able to get results in time for this paper, we expect to have results shortly that will be included in the final version of the paper.

10. REFERENCES

[1] F. Alexander and A. Garcia. Direct simulation Monte Carlo. *Computers in Physics*, 11:588, 1997.

[2] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, 1989.

[3] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[4] M. J. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man., and Cybernetics*, 21:1278–1286, September 1991.

[5] G. Bryan and M. L. Norman. Simulation x-ray clusters with adaptive mesh refinement. In *Proceedings of the 12th Kingston meeting on Theoretical Astrophysics: Computational Astrophysics (ASP Conference Series, 123)*, 1997. eds. D. A. Clarke and M. J. West, p. 363.

[6] A. C. Calder, B. C. Curtis, L. J. Dursi, B. Fryxell, G. Henry, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, H. M. Tufo, J. W. Truran, and M. Zingale. High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors. In *Proceedings of SC00*, 2000.

[7] M. R. Dorr, F. X. Garaizar, and J. A. F. Hittinger. Simulation of laser plasma filamentation using adaptive mesh refinement. Technical Report UCRL-JC-138330, LLNL, 2001. Submitted to J. Comput. Phys.

[8] A. Garcia, J. Bell, W. Crutchfield, and B. Alder. Adaptive mesh and algorithm refinement using direct simulation Monte Carlo. *Journal of Computational Physics*, 154:134–155, 1999.

[9] Gutman. Use of morton space-filling curve for load balance. *Dr. Dobb's Journal*, pages 115–121, July 1999.

[10] R. Hornung. A hybrid model for gas dynamics: Continuum-DSMC with AMR. In *First SIAM Conference on Computational Science and Engineering, Washington D.C.*, Sept 21–23 2000. Also available as Lawrence Livermore National Laboratory technical report UCRL-VG-139774.

[11] R. D. Hornung and S. R. Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the First Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. See `http://www.llnl.gov/CASC/SAMRAI`.

[12] R. D. Hornung and S. R. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency: Theory and Practice*, 2001. (to appear).

[13] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott. Large scale parallel structured amr calculations using the samrai framework. In *Proceedings of the SC01, Conference on High Performance Networking and Computing, Denver, CO*, November 10–16 2001. See `www.sc2001.org/papers/pap.pap146.pdf`.